A Parallel Algorithm for Finding Fill in a Sparse Symmetric Matrix

John R. Gilbert*
Hjálmtýr Hafsteinsson†

TR 86-789

October 1986

Department of Computer Science
Cornell University
Ithaca, NY 14853

* This author's research was partially supported by the National Science Foundation under grant DCR-84-51385.
† Fulbright Scholar, research partially supported by IBM Corporation under contract 573616

A Parallel Algorithm for Finding Fill in a Sparse Symmetric Matrix

John R. Gilbert*

Hjálmtýr Hafsteinsson†

Computer Science Department
Cornell University
Ithaca, New York 14853

October 1986

Abstract

We describe a parallel algorithm for finding the fill that occurs when a sparse symmetric positive definite matrix A is factored into its Cholesky factor L. The algorithm is in two steps: First we determine the elimination forest F for A. Then from F and A we compute the fill. The algorithm takes $O(\log^2 n)$ time, using $m+n$ processors to find the elimination forest and $m^* + n$ processors to find the fill.

* This author's research was partially supported by the National Science Foundation under grant DCR-84-51385

† Fulbright Scholar, research partially supported by IBM Corporation under contract 573616

1. Introduction

In this paper we describe a parallel algorithm for finding the fill that occurs when a sparse symmetric positive definite matrix A is factored into its Cholesky factor L, which is the unique lower triangular matrix with positive diagonal elements such that $A = LL^T$. In general, the nonzero positions in L are the nonzero positions in A plus some extra positions; these extra positions are called *fill*. Computing fill is the first step of efficient algorithms for Cholesky factorization both serially [2] and in parallel [9]. The algorithm is in two steps: First we determine the elimination forest F (see definition below) for A. Then from F and A we compute the fill. Each step runs in $O(\log^2 n)$ time. The first step uses $n + m$ processors and the second uses $n + m^*$ processors, where n is the dimension of the matrix, m is the number of nonzeros in A, and m^* is the number of nonzeros in L. The algorithm uses space linear in the number of processors.

Rose et al. [5] give an $O(m^*)$ time sequential algorithm to find fill. There is also a straightforward parallel algorithm using n^3 processors and time $O(\log^2 n)$ for finding the fill. Our algorithm reduces the number of processors needed and brings the product of processors and time down to within a factor of $\log^2 n$ off the optimal. Zmijewski and Gilbert [9] give an algorithm to compute elimination forests on a message-passing multiprocessor in time $O((m/p)a(m/n, n) + na(n, n)\log p)$, where $a(m, n)$ is related to the inverse of Ackerman's function and p is the number of processors. That algorithm is efficient when $p \ll n$ and short messages are nearly as expensive as long messages, but our algorithm is asymptotically superior when p is large compared to n.

The model of computation we will use is a CRCW with shared memory. Write conflicts are resolved by having an arbitrary processor succeed. We also assume that the model has the property that each processor is able to spawn another processor when needed.

The symmetric matrix A can be viewed as an adjacency matrix of an undirected graph G(A). Each row and column of A corresponds to a vertex in G(A) and each nonzero of A corresponds to an edge in G(A). In the following we will often talk about finding fill edges in G(A), instead of finding the fill of A. Thus we wish to compute the *filled graph*, which is $G(L + L^T)$.

Sequentially the fill is found by playing the elimination game on G(A) [2]. It consists of eliminating the vertices in order; when a vertex i is eliminated we add a fill edge between every pair of neighbors of i that are not already eliminated. An equivalent way of defining the fill, and one that will be more useful for us, is to say that there will be a fill edge between vertices i and j of G(A) if and only if there is a path between i and j using only vertices numbered lower than i and j.

It is useful to know both the elimination forest and the fill for the symbolic part of Cholesky factorization. The elimination forest is also used in the numerical part and it is a useful model of parallelism in the Cholesky factorization. The elimination forest has been studied by Schreiber [6] and Liu [3]. The best sequential algorithm for finding the elimination forest takes $O(m\alpha(m\ n))$ time [3]. Our algorithm for finding the elimination forest (i.e. the first part of the fill algorithm) takes $O(\log^2 n)$ time using $m+n$ processors.

The rest of the paper is organized as follows. Section 2 describes the algorithm to find the elimination forest. Section 3 describes the algorithm to find the fill from the forest. In sections 2 and 3 we assume the matrix is a dense two dimensional array, so adding an edge or testing for the presence of an edge takes constant time. In section 4 we modify the algorithms to run in space linear in the number of processors.

2.1 Finding the elimination forest

The first part of the algorithm involves finding the elimination forest F. Let's first define an elimination forest.

Definition:

Let $A = (a_{ij})$ be an $n \times n$ symmetric positive definite matrix and let $L = (l_{ij})$ be its Cholesky factor. The *elimination forest* for A is a directed forest $F = (V, E)$ where $V = \{1, 2, \ldots, n\}$ and $(i, j) \in E$ if and only if

$$j = \min \{ k \mid l_{ki} \neq 0 \text{ and } k > i \}$$

The graph F is a forest because j is uniquely defined for each i. Thus i is a child of j in F. We will use a divide-and-conquer approach to finding F. The algorithm is given below as Algorithm 1.

At some level in the algorithm we have the situation in figure 1. The subgraph C consists of all the vertices from h through k and the edges from G(A) that have both their endpoints in that set. If C contains more than one vertex we split it up so that C_1 is the subgraph consisting of the vertices from h through l and the edges within that set and C_2 is the subgraph of vertices from $l+1$ through k and associated edges, where l is halfway between h and k.

Before we recur on C_1 and C_2 we want to add to C_2 enough fill edges that result from eliminating the vertices of C_1 so that the resulting graph, $\hat{C}_2$, will have an elimination forest that is an induced subgraph of the correct elimination forest for C. For instance if (i, j) and (p, q) are edges between C_1 and C_2 (i.e. $i, p \in C_1$ and $j, q \in C_2$) and there is a path in C_1 between i and p then (j, q) is a fill edge in $\hat{C}_2$ (see figure 2.).

We won't add all those edges, only the ones between the smallest vertex in C_2 adjacent to each connected component of C_1 and all other vertices of C_2 adjacent to the same component of C_1. That

func findelim(C : graph) : forest
  {C is the subgraph consisting of the vertices h through k and its edges}
  if $h = k$ **then**
    findelim := $\{(\{h\}, \varnothing)\}$
  else
    $l := \lfloor (h+k)/2 \rfloor$
    $C_1 :=$ the subgraph consisting of vertices h through l
    $C_2 :=$ the subgraph consisting of vertices $l+1$ through k
    $\hat{C}_2 :=$ addfilltoC$_2$(C_1, C_2)
    pardo
      $F_1 :=$ findelim(C_1)
      $F_2 :=$ findelim($\hat{C}_2$)
    od
    findelim := merge(F_1, F_2)
  fi
end

Algorithm 1

Figure 1

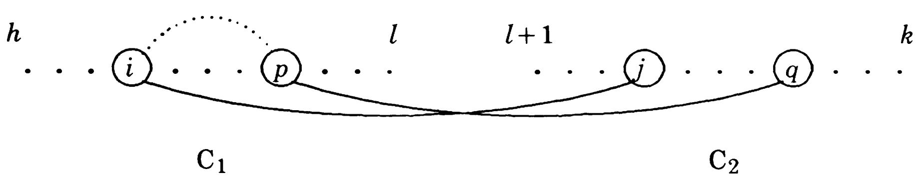

Figure 2

will be shown to be enough to make $\hat{C}_2$ have the elimination forest that is an induced subgraph of the correct one. When these fill edges have been added to $\hat{C}_2$ we can recur on $\hat{C}_2$. We recur on C_1 and $\hat{C}_2$

in parallel, coming back up with elimination forests for each of them. These two forests are then merged, giving us an elimination forest for C.

There are now two things we have to show how to do: adding the fill edges caused by C_1 to C_2 and merging the elimination forests for C_1 and $\hat{C}_2$.

2.2 Adding fill edges to C_2

The key idea in adding the fill edges to C_2 is the fact that the vertices of C_2 that are adjacent to the same connected component of C_1 will eventually form a clique of fill edges in $\hat{C}_2$. However we don't have to add all those edges to C_2 to get $\hat{C}_2$, only enough so that $\hat{C}_2$ will have the right elimination forest.

To exploit this idea we first form a new graph C_1'. Its vertex set $V(C_1')$ is $V(C_1) \cup \{v_u \mid$ there is an edge (u, v) in C with $u \in V(C_1)$ and $v \in V(C_2)\}$. The edge set $E(C_1')$ is $E(C_1) \cup \{ (u, v_u) \mid u \in V(C_1)\}$. Therefore C_1' has all the vertices and edges of C_1 and also a vertex for each edge between C_1 and C_2. Those vertices will only have degree 1 in C_1'. In other words, we split up each vertex of C_2 that has an edge coming from C_1 into as many vertices as the number of edges coming from C_1.

We now use Shiloach and Vishkin's connected components algorithm [7] on C_1'. It runs in $O(\log(|V(C_1')|))$ time using $|E(C_1')| + |V(C_1')|$ processors. Note that whenever some v_u-vertices of C_1' are in the same connected component, their associated v-vertices of $\hat{C}_2$ will form a clique of fill edges in the final filled graph.

Next we want to find the lowest numbered vertex in C_2 adjacent to each connected component of C_1. To do that we allocate an array MIN of size $|V(C_1')|$ in memory. Each element of the array consists of $\log(|V(C_1)|)$ bits. All the elements of MIN are initialized to contain the bit sequence $11\cdots 1$. We also assign a processor P_{vu} to each v_u-vertex. Let k be the component number of a v_u-vertex. Its assigned processor P_{vu} looks at the first (most significant) bit of the associated v-vertex of $\hat{C}_2$. If it is 0 then P_{vu} writes 0 in the first bit position of MIN[k]. Each processor then reads the first bit of MIN[k] and if someone actually put 0 there, then those processors that didn't try themselves to put 0 will stop. The rest then do the same for the second bit of MIN[k] and so on.

After $\log(|V(C_1)|)$ iterations of this, the processors still running for a particular k all have the same associated v-vertex in $\hat{C}_2$ and the bit sequence in MIN[k] is the binary representation of that v. Thus that v is the minimum vextex in $\hat{C}_2$ adjacent to component k of C_1'. Finally all the processors that stopped during the $\log(|V(C_1)|)$ iterations look at the number in MIN[k], for their respective k's, and add an edge between that v-vertex and their associated v-vertex in $\hat{C}_2$. What we have now is that for each connected component k of C_1, the smallest vertex in C_2 that is adjacent to k has an edge to all the other vertices in C_2 that are also adjacent to k.

The total time for adding these fill edges to $\hat{C}_2$ is $O(\log(|V(C_1')|)) = O(\log(|V(C_1)|))$ and the number of processors used is $|E(C_1')| + |V(C_1')| \leq |E(C)| + |V(C)|$.

2.3 Merging elimination forests

We now have elimination forests for C_1 and $\hat{C}_2$ and we want to merge them to get an elimination forest for C. We will assume that we not only have the elimination forests F_1 and F_2 but also associated forests E_1 and E_2, the path compressed versions of F_1 and F_2 (see figure 3).

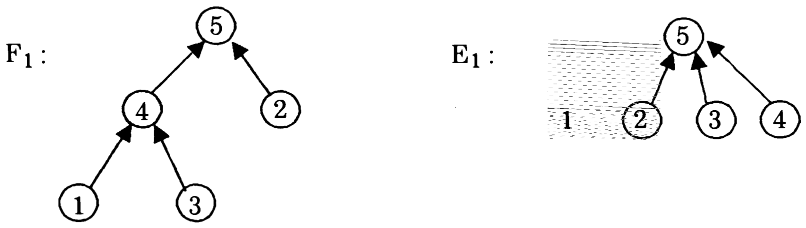

Figure 3

If (i, j) is an edge between C_1 and $\hat{C}_2$ (with $i \in C_1$ and $j \in \hat{C}_2$) then we want to hook the root of i's elimination tree onto j. However if there is another edge (p, q) between C_1 and $\hat{C}_2$ with p in the same tree as i and with $q < j$ then we want instead to hook i's tree onto q. Thus we want a tree T in C_1 to be hooked onto the smallest node in $\hat{C}_2$ that is adjacent to a node in T.

We allocate an array MIN of size $|V(C_1)|$ in memory and initialize its elements to all contain the bit sequence $11 \cdots 1$. We then assign a processor P_{ij} to each edge between C_1 and $\hat{C}_2$. Let k be the root of i's elimination tree in C_1. Then P_{ij} puts 0 into the first (most significant) bit position of MIN[k] if the first bit of j is 0. All the processors then read the first bit of MIN[k] and if it is 0 then those processors that didn't try to put 0 there will stop. If the bit is 1 then all the processors go on to the second bit of MIN[k]. Those that did put 0, if the bit was 0, also go on to the second bit of MIN[k] and so on.

After $\log(|V(C_1)|)$ iterations of this there will be at most one processor per element of MIN still running. If that processor is P_{ij} for MIN[k] then the bit sequence in MIN[k] is the binary representation of j. That means that j is the lowest numbered vertex in $\hat{C}_2$ adjacent to i's elimination tree. Thus P_{ij} can hook k up to j and also hook i's tree in E_1 to j's tree in E_2. This takes $\log(|V(C_1)|)$

time using as many processors as there are edges between C_1 and $\hat{C}_2$. We then need $|V(C)|$ processors to do one step of path compression on the, now combined, forests E_1 and E_2.

2.4 Timing analysis for finding elimination forests

In the divide-and-conquer algorithm there are $\log n$ levels, each involving subgraphs with half as many vertices as the one before. On the way down we add fill edges, caused by C_1, to C_2. That takes time $O(\log(|V(C_1)|))$. In the beginning $|V(C_1)|$ is $n/2$ and it is halved at each level. Thus the total time on the way down is $O(\log^2 n)$. The number of processors used is $|E(C)| + |V(C)|$ at each level. Initially $|E(C)| = |E(G)|$ and $|V(C)| = |V(G)|$. At each level the sum of all the $|E(C)|$'s on that level is at most $|E(G)|$ and the sum of the $|V(C)|$'s on that level is no more than $|V(G)|$. Hence the total number of processors used going down the recursion is of the order $|E(G)| + |V(G)|$.

In coming up the recursion we use $\log(|V(C_1)|)$ time at each level to merge the elimination trees with no more than $|V(G)| + |E(G)|$ processors at each level.

Combining these numbers we find that the time for finding the elimination forest for G is $O(\log^2 n)$, using $m + n$ processors.

2.5 Proof of correctness

To prove the correctness of the algorithm we first need the following lemma.

Lemma 1

If the algorithm produces correct elimination forests F_1 and F_2 for C_1 and $\hat{C}_2$,then they are induced subgraphs of the correct elimination forest for G.

Proof:

It is obvious that F_1 is an induced subgraph of the correct forest, since we don't have to worry about the higher numbered vertices when we produce F_1. For F_2 to be an induced subgraph of the correct forest we need $\hat{C}_2$ to have the same elimination forest as the $(n/2)$-th elimination graph of G. The elimination graph [2] of G is the graph that results if we start with G and at each step i of the Cholesky factorization of A add the fill edges and delete vertex i. After k such steps we get the k-th elimination graph of G, G_k. By Theorem 5.1.3 in [2] there is a fill edge in $G_{n/2}$ between two vertices of $\hat{C}_2$ iff there is a path (possibly empty) between them that only goes through vertices in C_1. Obviously, the elimination forest of $G_{n/2}$ is an induced subgraph of the correct one. We want to show that $\hat{C}_2$ has the same elimination forest as $G_{n/2}$. If the algorithm above were to make all the vertices of C_2 that are adjacent to the same connected component of C_1 into a clique, then the resulting graph would be $G_{n/2}$. However the algorithm only puts edges between the smallest vertex in C_2 adjacent to a component in C_1 and the other vertices in C_2 adjacent to the same component. But when this smallest vertex is eliminated that will make all the other vertices into a clique, so the effect is the same as making them all into a clique initially. Thus $\hat{C}_2$ has the same elimination forest as $G_{n/2}$, which is an induced subgraph of the correct forest.

Theorem 1

Let F^{ALG} be the forest generated by the algorithm for the graph G and let F be the correct elimination forest for G. Then $(i,j) \in F^{ALG}$ if and only if $(i,j) \in F$.

Proof:

We use induction on the levels in the algorithm.

Base case: If there is only one vertex then the theorem holds trivially, since there is no edge in F and the algorithm will not produce any edge.

Ind. step: We assume that F_1 and F_2 are the correct elimination forest for C_1 and $\hat{C}_2$ respectively. By Lemma 1 we know that F_1 and F_2 are induced subgraphs of F. That means that whenever vertex i has a parent in F_1 or F_2 then it has the same parent in F. But if i doesn't have a parent in F_1 or F_2 and has one in F then i must be in C_1 and its parent in $\hat{C}_2$. Thus we need to consider the vertices in C_1 that don't have any parents in F_1.

Let $i \in C_1$ have no parent in F_1. Then the algorithm will set as the parent of i the lowest numbered vertex j in $\hat{C}_2$ that has an edge to some vertex h in i's elimination tree in F_1. But each of those edges (k, h) will cause l_{ki} to be nonzero. Furthermore this $j \in \hat{C}_2$ that the algorithm picked is the smallest vertex greater than i with $l_{ji} \neq 0$, because it is the smallest one in $\hat{C}_2$ and there is no vertex in C_1 that satisfies these conditions, or else i would have had a parent in C_1. Thus the j picked by the algorithm is the one the definition of the elimination forest says is i's parent.

Let $(i,j) \in F$ with $i \in C_1$ and $j \in C_2$. Then by definition j is the smallest vertex in C_2 with $l_{ji} \neq 0$. This is the same as saying that j is the smallest vertex in C_2 that has an edge to i's elimination tree. But that is just the j that the algorithm selects, so $(i,j) \in F^{ALG}$.

Hence $F^{ALG} = F$.

The theorem tells us that the algorithm correctly computes the elimination forest for G.

3.1 Finding the fill

We now have the elimination forest and want to use it to find the fill that occurs in the graph (i.e. matrix). Three properties of elimination trees are useful here. Elimination trees are heap ordered, so that all the vertices are higher numbered than their children in the tree. Also, if (i,j) is an original graph edge and $i < j$ then j is an ancestor of i in the elimination tree. Finally, for all vertices k on the path from i through j in the tree, there is a fill edge (k,j) [3].

The algorithm consists of two stages. In the first stage we associate a processor P_{ij} with each original graph edge (i,j). If the distance from i to j is an integer power of 2 then P_{ij} spawns another processor, puts it on i, and is done with the first stage. Otherwise it spawns a processor and puts it on the vertex p_1 whose distance from j is the largest integer power of 2 less than the distance between i and j (see figure 4). Now, if the distance from i to p_1 is an integer power of 2 then P_{ij} spawns a

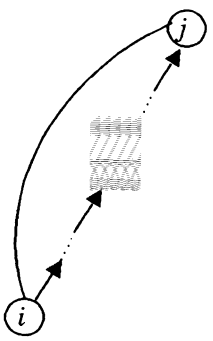

Figure 4

processor, puts it on i and is finished with the first stage. Otherwise it spawns a processor and puts it on the vertex p_2 whose distance from p_1 is the largest integer power of 2 less than the distance between i and p_1. This goes on until P_{ij} puts a processor on i. Thus the distance between each of j, p_1, p_2, ... is a strictly decreasing integer power of 2. Each of the processors on these vertices will, in the second stage, be responsible for covering the range from the vertex they are on through the vertex that the next processor is on. For example, the processor on p_2 will attempt to add a fill edge (v,j) for all vertices v on the path from p_2 to p_1.

In the second stage all the processors spawned by a processor P_{ij} in the first stage start at the same time and add a fill edge between the vertex p_k that they are on and j. Then the processors find the vertex u that is halfway between p_k and p_{k-1}, spawn a processor and put it on u. The new processor then enters the fill edge (u,j). This is then done recursively on both processors. Note that the number of vertices in the range from p_k to p_{k-1} is an integer power of 2, so we will be generating a complete binary tree of processor invocations.

The elimination forest is first preprocessed. For each tree in the forest we find a preorder numbering of the nodes. In addition we find the level number (i.e. depth) of each node and set up an array K for each level of the tree. This array K contains all the vertices on this level of the tree, ordered by preorder number. This preprocessing takes time $O(\log n)$ using n processors [8].

Since we know the level numbers of i and j, in stage one we can calculate the level number of the vertex p_1. We then use binary search on the array $K(\mathrm{lev}(p_1))$ to find the actual vertex p_1. It is the vertex in $K(\mathrm{lev}(p_1))$ whose preorder number is closest to preorder(i) from below. This will take at most $O(\log n)$ time for each vertex p_k. The maximum number of processors that P_{ij} has to spawn is $\log n$. Thus the total time for stage one is $O(\log^2 n)$.

```
proc stage1(i, j : vertex)
    k := j
    d := lev(i) - lev(j)
    while d ≠ 0 do
        l := 2^⌊log d⌋
        p := binsearch(K(l + lev(k)), preorder(i))
        spawnproc(j, p, k)
        k := p
        d := d - l
    od
end
```

Algorithm 2

Stage two starts with all the processors spawned in stage one with spawnproc(j, p, k) simultaneously executing the following statements:

```
addfilledge(p, j)
stage2(j, p, k)
```

If two processors try to enter the same fill edge at the same time then an arbitrary one succeeds. Also, if there is already a fill edge there then the processor does not succeed. If the processor doesn't succeed in entering the fill edge then it will not spawn another processor, but will continue recurring on the path between i and p. If two processors try to add the same fill edge (p,j) at the same time then their range from p to k is the same. Their lower range (from i to p) might not be the same, though. The same thing applies if a processor tries to add a fill edge that is already there.

```
proc stage2(j, i, k : vertex)
    if (i ≠ k) then
        l := (lev(i) + lev(k))/2
        p := binsearch(K(l), preorder(i))
        addfilledge(p, j)
        if (success) then
            pardo
                stage2(j, i, p)
                stage2(j, p, k)
            od
        else
            stage2(j, i, p)
        fi
    fi
end
```

Algorithm 3

The time that stage two takes is no more than O(log$n$) for each level of recursion and there are no more than log$n$ levels. This gives us O($\log^2 n$) for the whole stage. A new processor is spawned only when an old processor successfully adds a fill edge, so the number of processors is the same as the number of fill edges plus the processors used for the preprocessing, i.e. $m^* + n$.

3.2 Proof of correctness

We will prove the correctness of Algorithms 2 and 3 in two parts.

Theorem 2

If the algorithm adds (v, j) then (v, j) is a fill edge (i.e. v is a descendant of j in a tree and there is a descendant i of v with an edge to j).

Proof:

If the algorithm adds (v, j) then the procedure fill was initially called because of some edge (i, j), but since each recursive call only involves vertices that are ancestors of i (or i itself) when we finally add the edge (v, j). v is an ancestor of i. Hence (v, j) is a fill edge according to the definition.

Lemma 2

Let P and P' be two processors executing stage2(j, i, k) and stage2(j, i', k') respectively.

(i) If P and P' try to add the vertex (v, j) at the same time then $k = k'$ and level(i) = level(i').

(ii) If P tries to add the edge (v, j) but does not succeed because P' had already added (v, j) then $k = k'$ and level(i) = level(i').

(iii) If P tries to add the edge (v, j) but does not succeed because (v, j) was added before stage two then v is equal to some p_l from stage one and $k = p_{l-1}$ from stage one.

Proof:

As a result of stage one, whenever a processor starts stage two the size of its range is an integer power of 2. In stage two it splits its range into two equal halves, so their size will also be an integer power of 2. Thus when a processor tries to add an edge (v, j), v is the upper midpoint of a range whose size is an integer power of 2. We want to prove that for all the processors trying to add (v, j), this size is the same. We claim that if the distance between v and j is a multiple of 2^l but not of 2^{l+1}, for some integer l, then the range is of size 2^{l+1}. Suppose $\hat{P}$ is starting starting stage two, assigned to vertex p_k, whose distance from j is a multiple of 2^l but not of 2^{l+1}. Then p_k's distance from p_{k-1} is also a multiple of 2^l and not of 2^{l+1}. It is also the largest integer power of 2 less than the distance from i to p_{k-1}, where (i, j) is the original graph edge. Thus the range that $\hat{P}$ has to cover is from p_k to p_{k-1} and it is of size 2^l and the midpoint of that range is a multiple of 2^{l-1} but not of 2^l.

If P and P' are two processors executing stage2(j, i, k) and stage2(j, i', k') respectively and they try to add the edge (v, j) at the same time then their ranges is the same size and v in the upper midpoint of those ranges. Hence $k = k'$ and level(i) = level(i'). Similarly, if P tries to add the edge (v, j) that P' has already added, their ranges are the same size and $k = k'$ and level(i) = level(i'). If P tries to add (v, j) but doesn't succeed because (v, j) was added before stage two then $v = p_l$, for some integer l. Then the upper range of P is the same as the range from p_l to p_{l-1}, so $k = p_{l-1}$.

Theorem 3

If (v, j) is a fill edge, then the algorithm will add (v, j).

Proof:

Since (v, j) is a fill edge there must be some descendant i of v in the tree that has an edge to j. Procedure stage1 will therefore be invoked on that edge. Assume that v is on the path between p_k and p_{k-1}. Thus procedure stage2 gets called with parameters j, p_k and p_{k-1}.

Each call stage2(j, i, k) attempts to cover the range from the lower bound i to the upper bound k. It splits the range into two parts, tries to add fill edge (p, j) and if successful makes a recursive call on the upper and lower half. If it doesn't succeed in adding the edge then it only goes on to cover the lower half of the range and we claim that the upper half will be covered by the processor that did add the fill edge (p, j).

There are two reasons why a processor P may not succeed in adding the fill edge. The first is that another processor P' added the edge at the same time. By lemma 2, whenever two processors try to add the same edge at the same time their upper ranges are the same. Since P' succeeded in adding the edge it will go on to cover both its upper and lower range, but its upper range is also from p through k so that range will be covered. The second reason for P not succeeding in adding the fill edge (p, j) is if a processor P" had already added the edge in a previous step. The upper range of P" is from p through k by lemma 2, so P" will cover that range. In any case the range from p through k will be covered. If P' or P" ever get to a point where they don't succeed in adding a fill edge we can apply the above argument again.

Thus, since the procedure stage2 will cover the whole range from p_k through p_{k-1}, it will add the fill edge (v, j).

4. Restricting the memory size

With $O(n^2)$ space, we could store the adjacency matrix of G, which allowed us to add a new edge or test for an edge in constant time. To restrict the amount of space to $O(m)$ for finding the tree and $O(m^*)$ for finding the filled graph, we keep a list of edges of G in memory, with a processor assigned to each edge. We also use $O(n)$ space and processors to store and manipulate the elimination tree.

The only tricky part is adding edges to the graph. Each time some edges are added, we will sort the list of old and new edges to remove duplicates and reassign processors to edges. The sort takes $O(\log n)$ time with n processors using Cole's parallel merge sort [1]

To analyze the overall running time, we need to go into a bit more detail. There are two places in the algorithm where edges are added to the graph.

Algorithm 1 must add edges to the graph in procedure addfilltoC$_2$. After the connected components of C_1 are found, the processors P_{uv} in each connected component find the smallest $v \in C_2$ adjacent to their component. Then each P_{uw} for that component with $w \neq v$ assigns itself the (possibly new) edge (v, w), forgetting edge (u, w). Then all the edges, old and new, are sorted to remove duplicates.

Notice that this procedure forgets some edges forever. The forgotten edges do not affect the elimination tree, however. For each edge (u, w) that is forgotten, there is an edge (x, v) with $x \in C_1$, $v \in C_2$, $v < w$, and x in the same component of C_1 as u. After the recursive call, then, the tree of F_1 containing x and u will have its root made a child of some vertex of C_2 less than or equal to v, whether or not edge (u, w) exists.

The other place edges are added to the graph is in Algorithm 3, when edges are added to the filled graph. Some edges may be added after each of the O(log$n$) steps of stage2. After each step, we sort the edges (new and old) to remove duplicates. The algorithm requires that each processor that tries to add an edge must find out whether it was successful. This can be done by associating with each fill edge the name of the processor that added it. After the sort, but before eliminating duplicates, all but the lowest-numbered of the processors that tried to add a given edge can be told that they failed.

In both Algorithms 1 and 3 the edge sort happens O(log$n$) times. Therefore the total time spent sorting is $O(\log^2 n)$, and the entire algorithm still takes $O(\log^2 n)$ time.

5. Conclusion

We have presented an algorithm to find the fill that occurs in the Cholesky factorization of a matrix. The algorithm has two parts. The first part is finding the elimination forest for the matrix, and the second is using the elimination forest to find the fill. Each part takes $O(\log^2 n)$ time with a number of processors equal to the size of its output.

There are a few open problems related to this work. One is whether this can be done faster, e.g. in O(log$n$) time, with the same number of processors. This doesn't seem likely, but if at all possible it would probably require a very different algorithm from the one presented above. Probably an easier problem is whether the fill can be found with only $m^*/\log n$ or even $m^*/\log^2 n$ processors.

Sparse Cholesky factorization consists of symbolic and numeric factorization. In the symbolic part we find out how much and where the fill will occur; our algorithm does this in low polylog time with $m^* + n$ processors. An interesting and possibly difficult question is whether the numeric part can be done in polylog time with as many processors as the number of arithmetic operations needed to factor the matrix.

References

[1] Cole, R. "Parallel merge sort." *Proc. 27th IEEE Symp. on Foundations of Computer Science* (1986), pp. 511-516.

[2] George, A. and J.W.H. Liu. *Computer Solutions of Large Sparse Positive Definite Systems.* Prentice-Hall, 1981.

[3] Liu, J.W.H.. "A compact row storage scheme for Cholesky factors using elimination trees." Tech. Report CS-84-02, York Univ., 1984.

[4] Preparata F.P.. "New parallel sorting schemes." IEEE Transactions on Computers C-27, 7 (1978), pp. 669-673.

[5] Rose, D.J., R.E. Tarjan and G.S. Lueker. "Algorithmic aspects of vertex elimination on graphs." *SIAM Journal on Computing*, 5 (1976), pp.266-283.

[6] Schreiber, R.. "A new implementaion of sparse Gaussian elimination." *ACM Transactions on Mathematical Software*, 8 (1982), pp. 256-276.

[7] Shiloach, Y. and U. Vishkin. "An O(log*n*) parallel connectivity algorithm." *J. Algorithms* 3 (1982), pp. 57-67.

[8] Tarjan, R.E. and U. Vishkin. "Finding biconnected components and computing tree functions in logarithmic parallel time." *Proc. 25th IEEE Symp. on Foundations of Computer Science* (1984), pp. 12-20.

[9] Zmijewski, E. and J.R. Gilbert. "A parallel algorithm for large sparse Cholesky factorization on a multiprocessor." Tech. Report CS-86-733, Cornell Univ., 1986.